Maxime Bernier - 1893115

Introduction

Dans ce laboratoire, nous avons créé les tests pour la méthode getBoardCards de la classe TrelloJsonCLient. Ce rapport expliquera nos démarches pour établir nos tests.

Démarches

testGetBoardCardsSuccess()

Premièrement, nous créons l’objet client de type TrelloJsonCLient avec le constructeur avec paramètres. Celle-ci nous sert à appeler la méthode à tester. Les paramètres passés (KEY, TOKEN, ENDPOINT) sont des variables globales pour le fichier de test. Ils sont de valeurs quelconques. Étant donné qu’aucune requête http n’est réellement survenue, c’est cela que nous devons simulée. En appelant la fonction getBoardCards(BOARD\_ID), un URL est créé. Or, celui-ci ne nous est pas accessible. Nous créons donc un mockURL qui respecte la même structure que celle de la fonction, soit ENDPOINT + "boards/" + BOARD\_ID + "/cards/". Ensuite, la fonction getJson(url) de l’attribut requestWrapper est appelé.

La première étape de cette méthode consiste à créer un objet de type CloseableHttpClient avec la méthode privée get(url,key,token). À l’intérieur de cette méthode, une autre méthode est appelée : getHttpClient(). Un problème survient : l’attribut httpClient n’est pas initialisé. On doit donc l’initialiser avec la méthode setHttpCLient. Or, le seul RequestWrapper est un attribut privé de notre client. Nous devons donc en créer un identique. Nous pouvons donc appeler la méthode setRequestWrapper pour s’assurer d’avoir le même RequestWrapper et la méthode setHttpClient tel que désiré. Le résultat étant un objet de type CloseableHttpClient, nous créons un mock de celui-ci. La création du httpGet ne nous cause aucun problème.

La ligne httpClient.execute(httpGet) est problématique. C’est ici que nous vérifierons qu’il s’agit bien d’un GET et du bon url. Voici la ligne de mock utilisée :

doReturn(mockCloseableHttpResponse).when(mockCloseableHttpClient).execute(aHttpGetRequestWithUriMatching(mockURI)); [1]

La méthode aHttpGetRequestWithUriMatching(mockURI) s’assure que l’URI est le même que celui de l’objet HttpGet est le même que celui passé en paramètre. Notre mockURI est constitué de la même façon que celui dans la méthode get() du RequestWrapper. De plus, la structure argThat( new HttpGetMatcher()) [1][2] s’assure de retourner le même type d’argument de HttpGetMatcher, soit HttpGet. Ainsi, nous sommes certain de la validité de l’URL et de qu’il s’agit d’un HttpGet. Nous forçons la sortie de ligne ci-haut à un mock de CloseableHttpResponse.

Nous revenons à la méthode getJson(). Nous sommes rendus à la condition if (response.getStatusLine().getStatusCode() >= HttpStatus.SC\_BAD\_REQUEST). Pour vérifier cette condition, un intermédiaire est instancié : StatusLine. Comme cela provient de la fonction getStatusLine appliqué à notre mockCloseableHttpResponse, nous allons forcer la sortie d’un mock StatusLine. Pour la même raison, on force la sortie 200, soit une requête acceptée [3], pour la méthode getStatus(). Nous ne respectons pas la condition alors nous passons par-dessus le bloc if.

Encore une fois, nous devons forcer un retour puisqu’une méthode est appelée sur notre mockCloseableHttpResponse. Nous devons donc créer l’entité HttpEntity. Celle-ci étant une interface, nous choisissons la classe dérivée StringEntity pour la simplicité de celle-ci. En effet, nous pouvons la construire avec un paramètre String. Ainsi, la conversion qui suit sera une simple formalité. Nous créons l’entité avec un string quelconque stringExpected. Nous sommes donc rendus au retour de la méthode à tester. Ainsi, après tous ses stubs et mocks, nous pouvons appeler notre méthode avec notre client créé précédemment. Nous utilisons un assertEquals entre la valeur expectedString et le retour de la méthode.

Pour le second test, la même logique s’applique. Rendu à la condition if, nous voulons toutefois rentrer dans le block if. Nous retournons ainsi la réponse 400, qui est une mauvaise requête. La première ligne du block if est équivalente aux deux lignes subséquentes à la fin du block if. Ainsi, nous pouvons réutiliser les mêmes stubs et mocks que le premier test. Ensuite, une exception de type TrelloException est lancée. Celle-ci fait appel au constructeur avec trois paramètres. Le premier et le troisème nous servirons à s’assurer que l’exception lancée est la bonne puisqu’il s’agit respectivement du code 400 et de l’expectedString. Il est à noter qu’une conversion implicite servient pour le code puisque celui-ci est défini comme un string dans le constructeur.

Pour la généralisation des méthodes http, il faudrait changer le HttpGetMatcher en HttpMatcher extends ArgumentMatcher<HttpUriRequest> et remplacer toutes les mentions de HttpGet par HttpUriRequest. Il s’agit d’une interface que chaque commande désirée (PUT, POST, GET, DELETE, PATCH) implémente.

[1] <https://stackoverflow.com/questions/13846837/using-multiple-argumentmatchers-on-the-same-mock?rq=1>

[2] <https://www.baeldung.com/mockito-argument-matchers>

[3] <https://httpstatuses.com/>

[4] <http://hc.apache.org/httpcomponents-client-ga/httpclient/apidocs/org/apache/http/client/methods/HttpUriRequest.html>

Autres sources mentionnées dans l’énoncé du tp et la documentation Apache.